4. Ваша первая 2D игра

В этой серии пошаговых уроков вы создадите свою первую полноценную 2D-игру с помощью Godot. К концу серии у вас будет своя простая, но законченная игра.
Вы узнаете, как работает редактор Godot, как структурировать проект и как построить 2D-игру.
Игра называется "Уверните от Крипов!". Ваш персонаж должен двигаться и избегать врагов как можно дольше.
Вы научитесь:

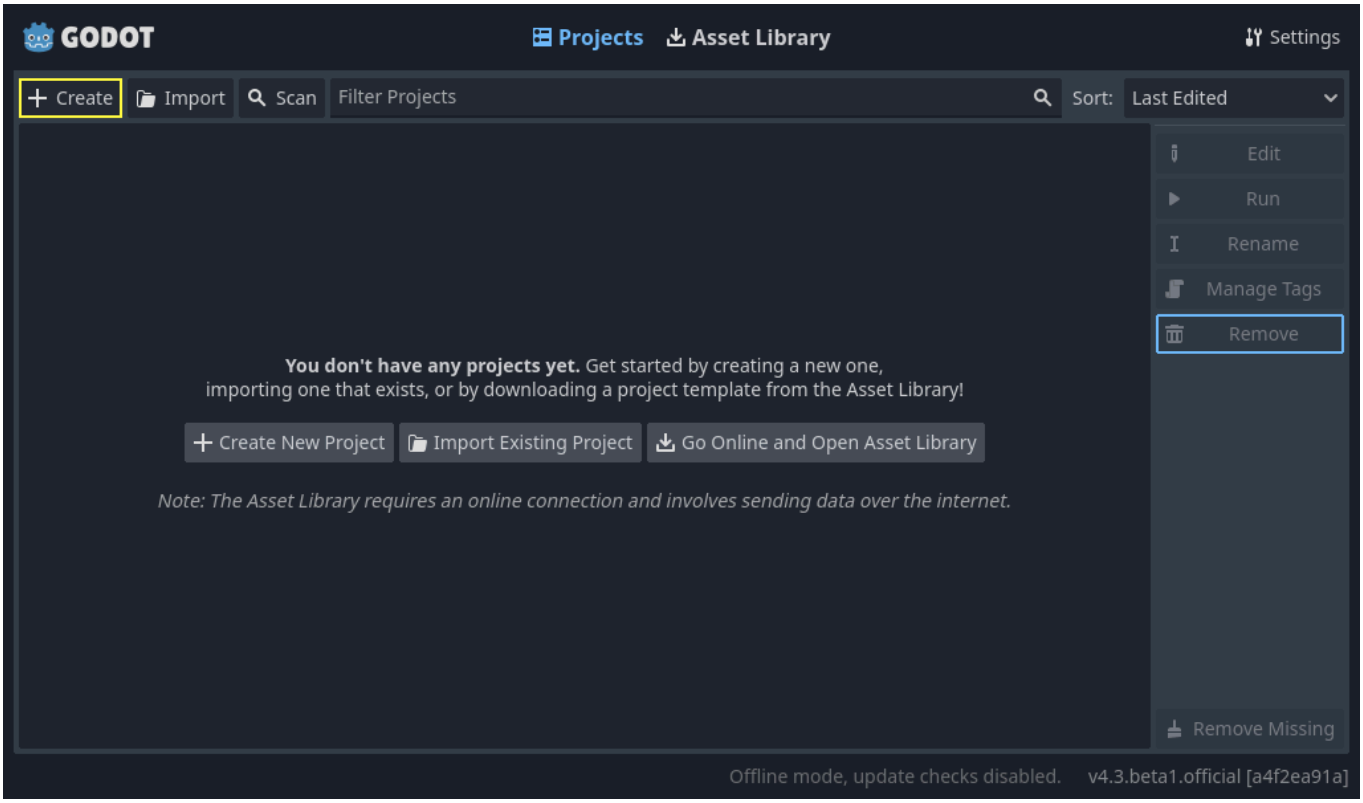
- Создавать законченную 2D игру в Godot editor.
- Структурировать простой проект игры.
- Перемещать персонажа игрока и изменять его спрайт.
- Порождать случайных врагов.
- Подсчитывать очки.

Почему стоит начинать с 2D?
Если вы новичок в разработке игр или не знакомы с Godot, мы рекомендуем начать с 2D-игр. Это позволит вам немного освоиться, прежде чем браться за 3D-игры, которые, как правило, сложнее создавать.

Настройка проекта

В короткой первой части, мы настроим и организуем проект.

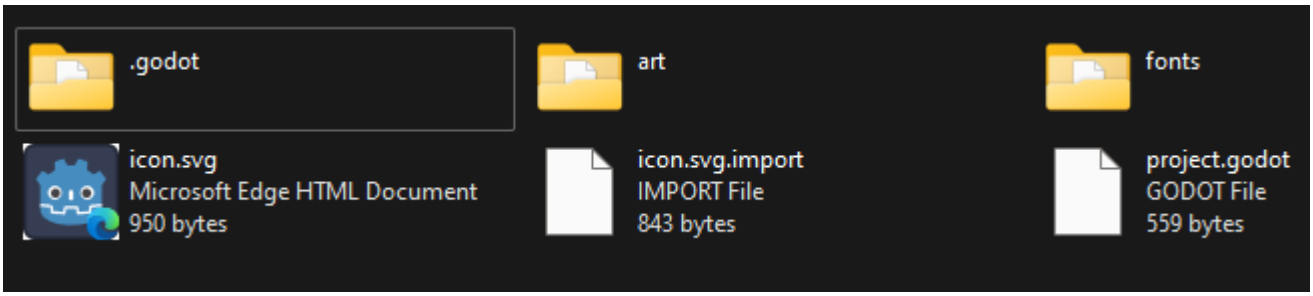
Запустите Godot и создайте новый проект.



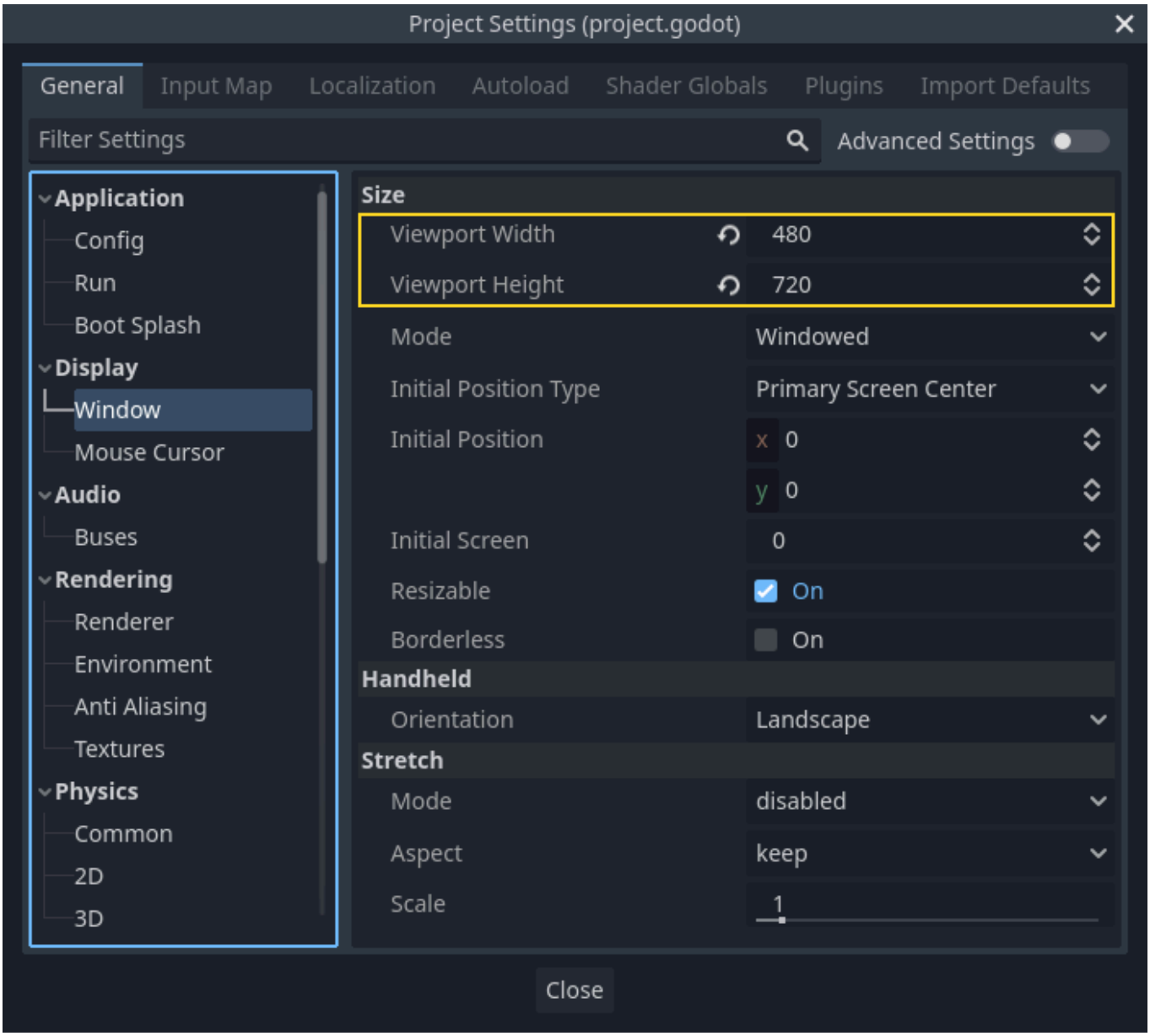
При создании нового проекта вам нужно только выбрать *Путь к проекту*. Остальные настройки вы можете не трогать.

> Архив содержит изображения и звуки, которые вы будете использовать для создания игры. Распакуйте архив и переместите папки `art/` и `fonts/` в папку вашего проекта.

Папка вашего проекта должна выглядеть следующим образом.

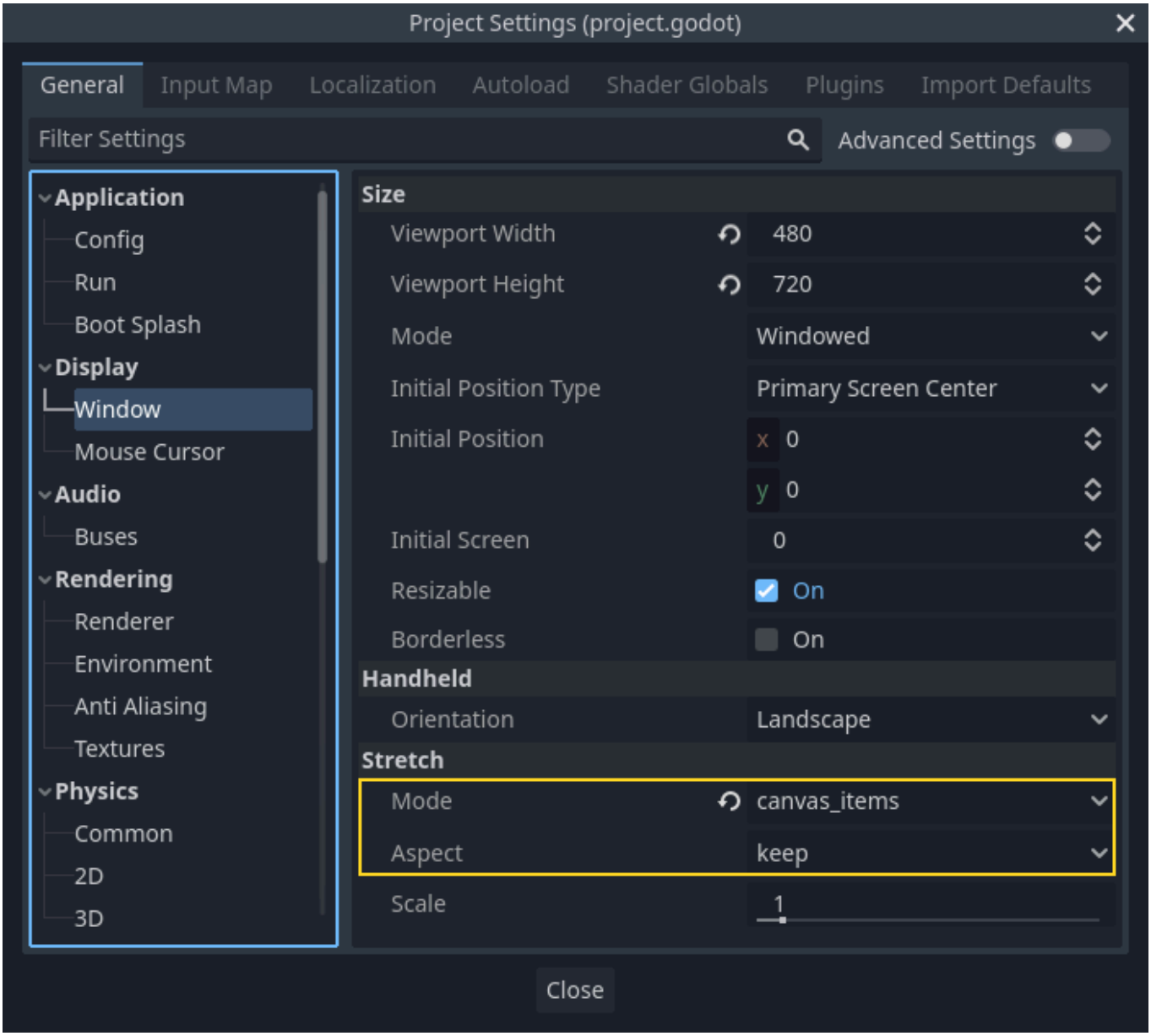


Эта игра будет в вертикальном режиме, поэтому нам нужно настроить размер окна игры. Нажмите на *Проект -> Настройки проекта*, чтобы открыть окно настроек проекта. В левой колонке откройте вкладку *Дисплей -> Окно*. Там установите "Ширину экрана" на `480` и "Высоту экрана" на `720`.



Кроме того, прокрутите раздел вниз и в опции **Растягивание** установите **Режим** на `canvas_items` и **Соотношение** на `keep`. Это

обеспечит стабильное масштабирование игры на экранах разного размера.



Организация Проекта

В этом проекте мы создадим 3 независимых сцены: `Player`, `Mob`, and `HUD`, которые мы образуем в сцену игры `Main`.

Для более удобной организации проекта рекомендуется для каждой сцены создать отдельную папку, в которой будет хранится сам файл сцены `.tscn` и все принадлежащие данной сцене ресурсы: скрипты, текстуры, шрифты и т.д.

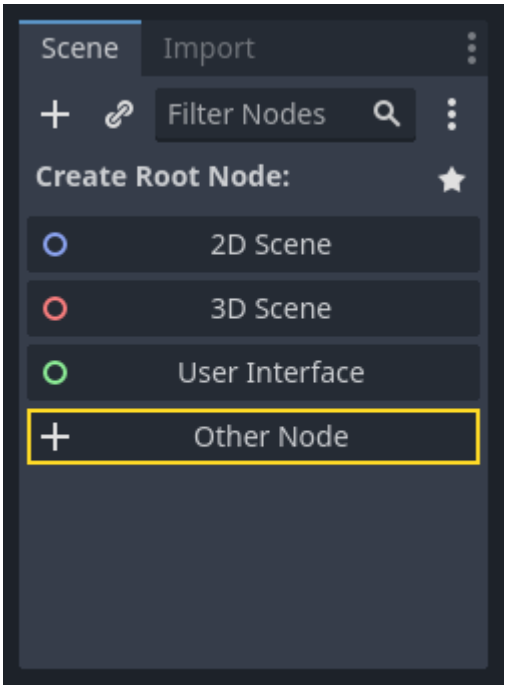
Создание сцены игрока

С установленными настройками проекта, мы можем начать работу над игроком.

Первая сцена будет определять объект `Player`. Одним из преимуществ создания отдельной сцены Player является то, что мы можем протестировать ее отдельно, даже до того, как создадим другие части игры.

Структура узла

Для начала нам нужно выбрать корневой узел для игрока. Как правило, корневой узел сцены должен отображать желаемую функциональность объекта - *чем* объект является. Нажмите кнопку "Другой узел" и добавьте узел Area2D в сцену.



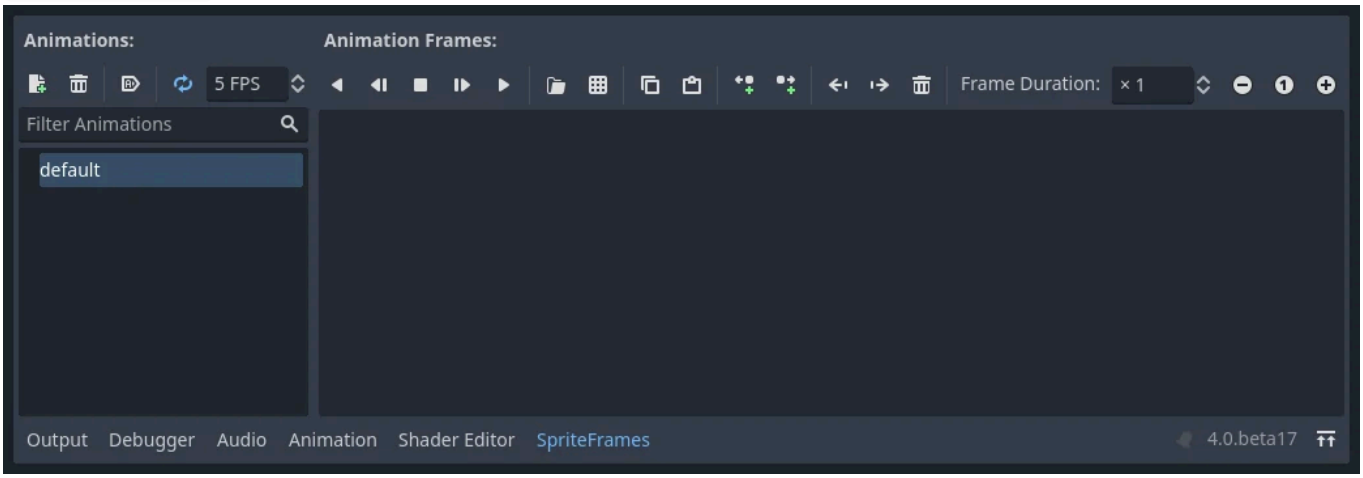
Godot отобразит значок предупреждения рядом с этим узлом в древе сцены. Пока что не обращайте внимание на это. Мы рассмотрим это позже.
С помощью `Area2D` мы можем обнаруживать объекты, которые перекрывают или сталкиваются с игроком. Измените имя узла на `Player`, дважды щёлкнув по нему. Теперь, когда мы установили корневой узел сцены, мы можем добавлять дочерние узлы, чтобы придать ему больше функциональности.
Прежде чем добавлять дочерние элементы в узел `Player`, мы хотим убедиться, что случайно не переместим или не изменим их размер, щелкнув по ним. Выберите узел и щелкните значок справа от замка. Его подсказка гласит: «Группирует выбранный узел с его дочерними элементами. Это приводит к выбору родительского элемента при щелчке по любому дочернему узлу в 2D- и 3D-виде».

Сохраните сцену. Нажмите "Сцена" -> "Сохранить сцену" в верхней панели или нажмите сочетание клавиш Ctrl + S
Для этого проекта мы будем следовать правилам именования Godot.

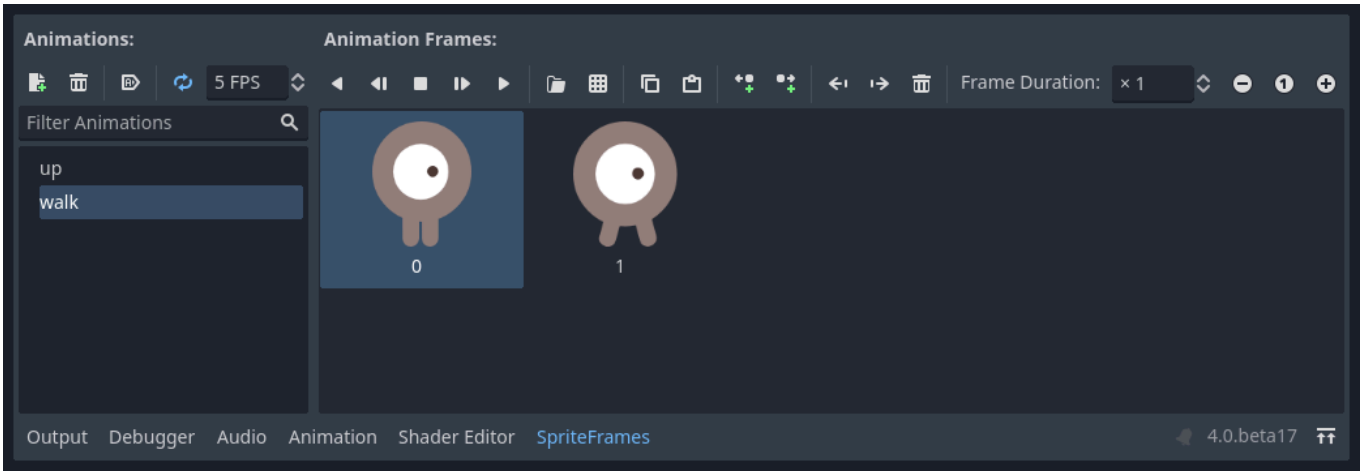
- **GDScript**: Классы (узлы) используют PascalCase, переменные и функции - snake_case, константы - ALL_CAPS

Анимация Спрайтов

Щелкните узел `Player` и добавьте дочерний узел `AnimatedSprite2D`. `AnimatedSprite2D` будет управлять внешним видом и анимацией для нашего игрока. Обратите внимание, что рядом с узлом есть предупреждающий символ. `AnimatedSprite2D` требует ресурс `SpriteFrames`, который представляет собой список анимаций, которые он может отображать. Чтобы создать его, найдите свойство `Sprite Frames` на вкладке `Animation` в `Inspector` и щелкните "[empty]" -> "New SpriteFrames". Щелкните только что созданный `SpriteFrames`, чтобы открыть панель "SpriteFrames":

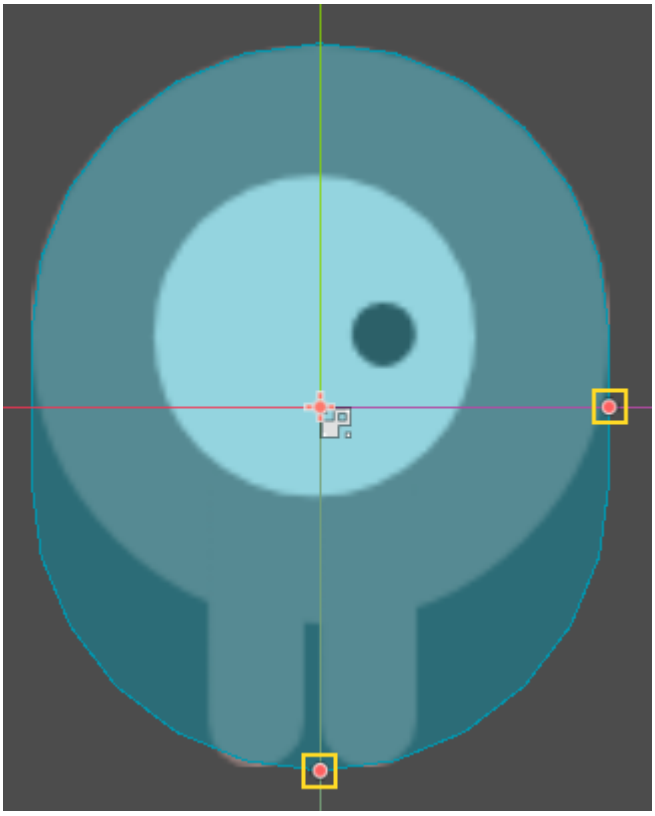


Слева находится список анимаций. Нажмите на "default" и переименуйте на "walk". Затем щелкните по кнопке "Добавить анимацию" для создания второй анимации с именем "up". Найдите изображения игрока на вкладке "Файловая система" - они находятся в папке "art", которую вы разархивировали ранее. Перетащите два изображения с названиями `playerGrey_up[1/2]` и `playerGrey_walk[1/2]` на сторону панели "Кадры анимации" для каждой анимации соответственно:

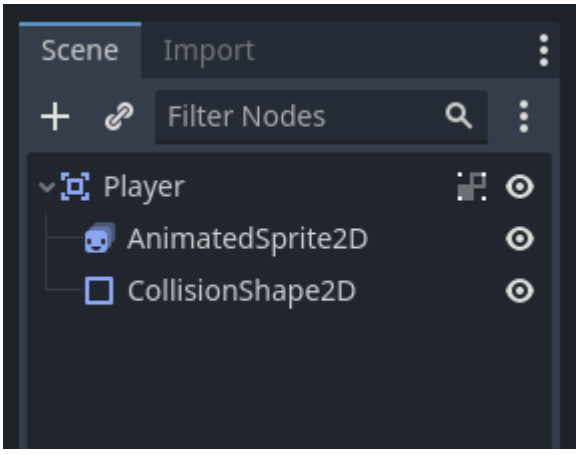


Изображения игрока слишком большие, поэтому нам надо их слегка уменьшить. Нажмите на узел `AnimatedSprite2D` и установите параметр `Scale` в `(0.5, 0.5)`. Вы можете найти его в Инспекторе под названием `Node2D`.

Наконец, добавьте CollisionShape2D в качестве дочернего узла `Player`. Это определит "хитбокс" игрока, или границы его области столкновения. Для нашего персонажа лучше всего подходит узел `CapsuleShape2D`, поэтому рядом с "Shape" в Инспекторе нажмите "[empty]" -> "New CapsuleShape2D". Используя две стрелки размера, измените размер фигуры, чтобы она покрывала спрайт:



Когда вы закончите, сцена `Player` будет выглядеть вот так:

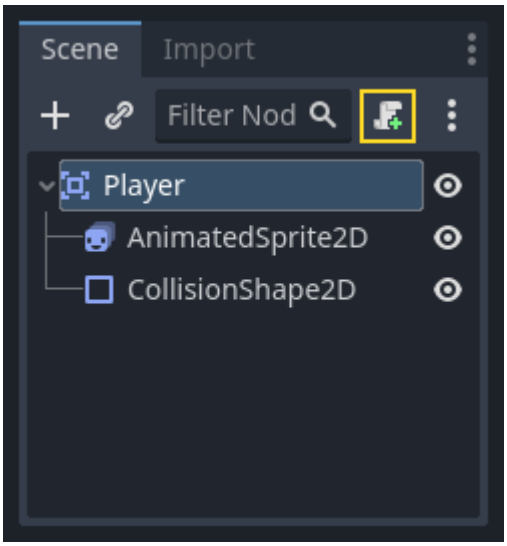


После этих изменений обязательно сохраните сцену еще раз.

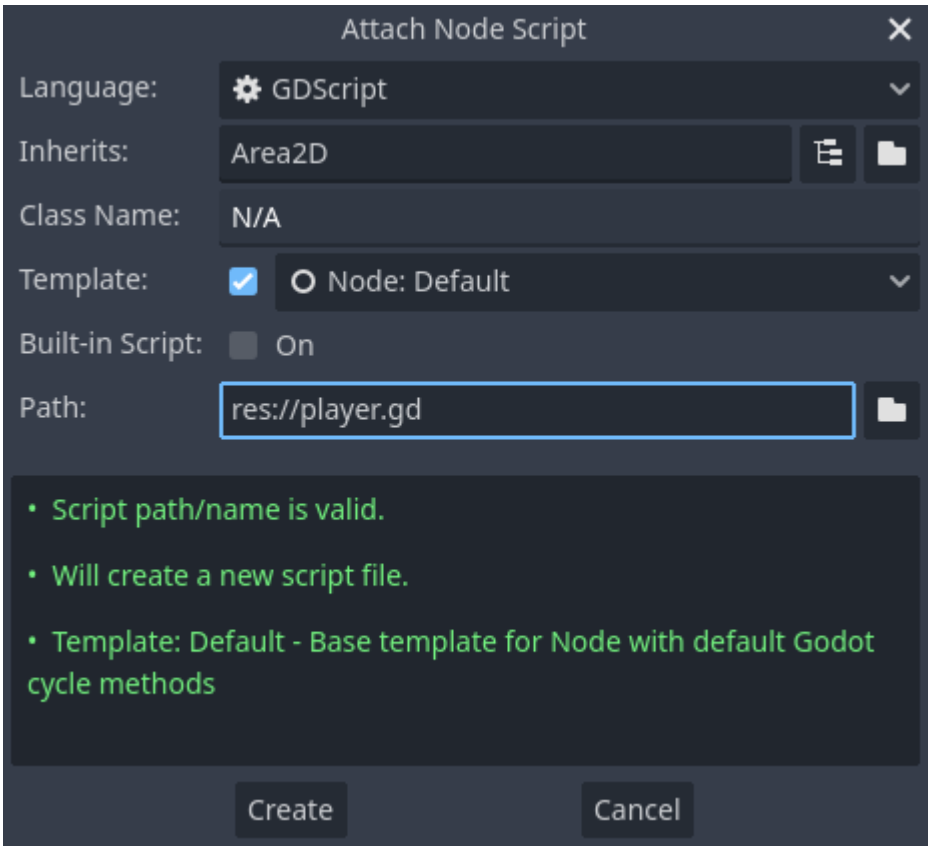
Движение Игрока

В этом уроке мы добавим передвижение игрока, анимацию и настроим всё для определения столкновений.

Для этого нужно добавить немного функционала, который мы не можем получить с помощью встроенных узлов, поэтому мы добавим скрипт. Нажмите на узел `Player` и нажмите кнопку "Attach Script" ("Прикрепить скрипт"):



В окне настроек скрипта вы можете оставить все настройки по умолчанию. Просто нажмите "Создать":

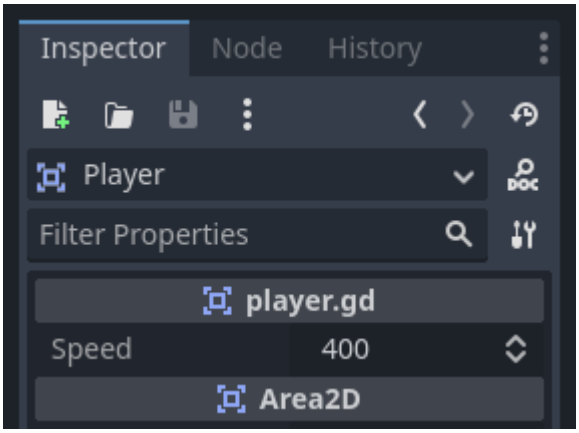


Начните с объявления переменных - членов, которые понадобятся этому объекту:

```
extends Area2D

@export var speed = 400 # How fast the player will move (pixels/sec).
var screen_size # Size of the game window.
```

Использование ключевого слова `export` для первой переменной `speed` позволяет нам задать ее значение в `Inspector`. Это может быть удобно для значений, которые вы хотите настраивать так же, как встроенные свойства узла. Щелкните узел `Player`, и вы увидите, что свойство теперь отображается в Inspector в новом разделе с именем скрипта. Помните, если вы измените значение здесь, оно переопределит значение, записанное в скрипте.



Ваш скрипт `player.gd` уже должен содержать функции `_ready()` и `_process()`. Если вы не выбрали шаблон по умолчанию, показанный выше, создайте эти функции, следуя уроку.

Функция `_ready ()` вызывается, когда узел появляется в дереве сцены, что является хорошим моментом для определения размера игрового окна:

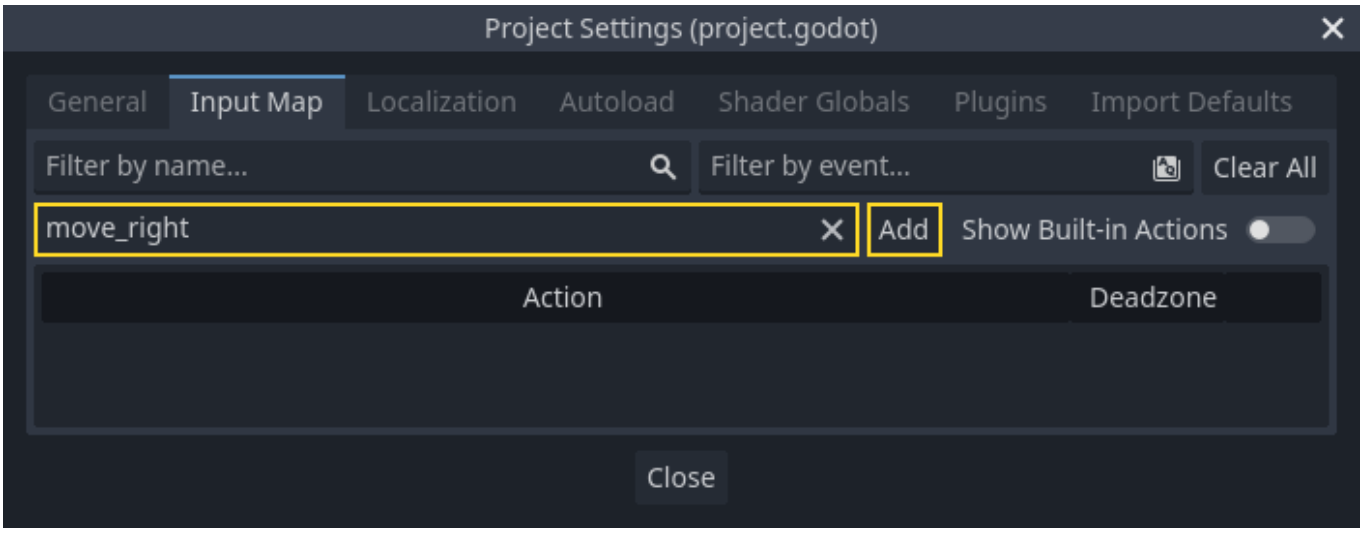
```
func _ready():
        screen_size = get_viewport_rect().size
```

Теперь мы можем использовать функцию `_process()` для определения того, что игрок будет делать. `_process()` вызывается каждый кадр, поэтому мы будем использовать ее для обновления состояния тех элементов нашей игры, которые будут часто изменяться. Для игрока, сделаем следующее:

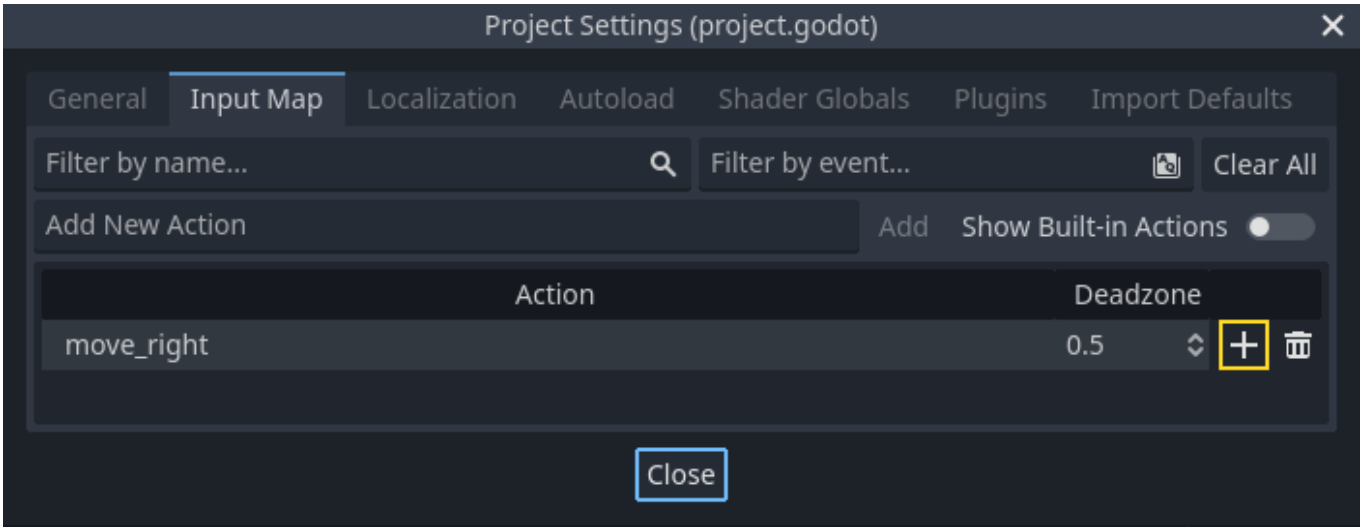
- Проверка ввода.
- Перемещение в заданном направлении.
- Воспроизвести соответствующую анимацию.

Во-первых, мы должны проверить ввод — нажимает ли игрок на клавишу? Для нашей игры нужно проверять 4 направления ввода. Функции ввода определяются в Настройках проекта во вкладке "Список действий". В ней можно определять отдельные события и назначать им различные клавиши, события мыши или другой ввод. В нашей игре мы присвоим клавиши стрелок для четырёх направлений.

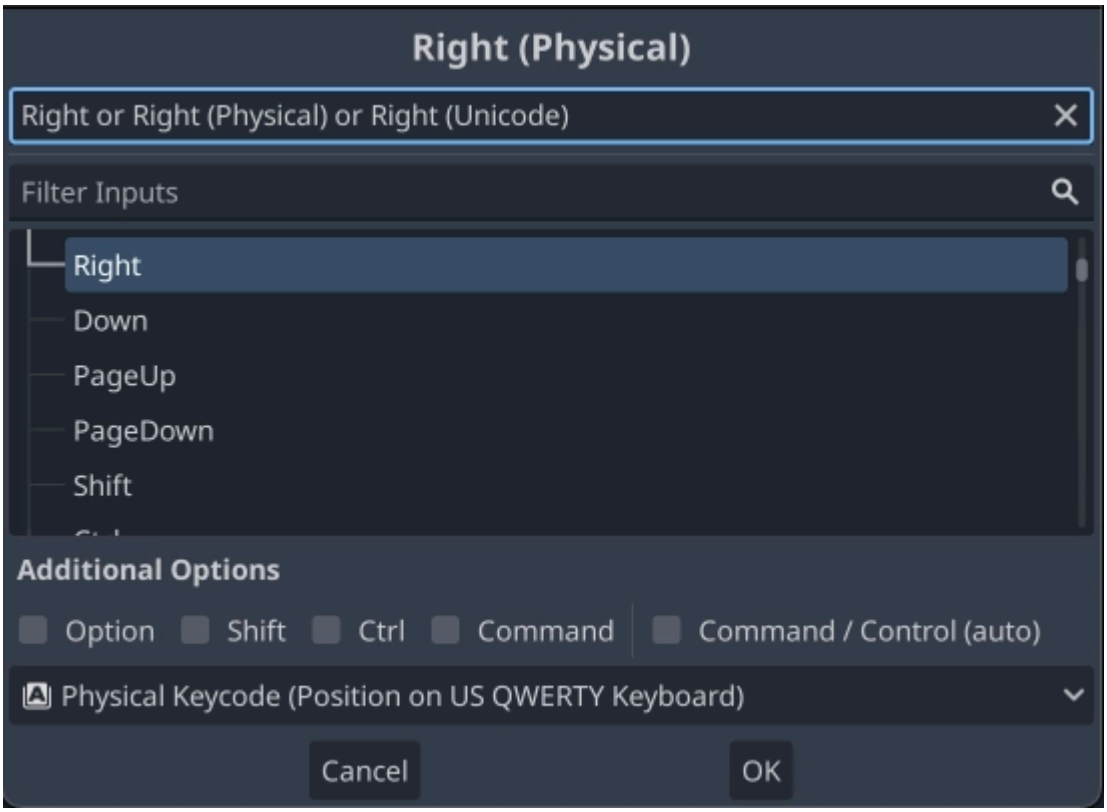
Нажмите на *Проект -> Настройки проекта*, чтобы открыть окно настроек проекта, и нажмите вверху на вкладку *Input Map*. Введите "move_right" в верхней панели и нажмите на кнопку "Добавить", чтобы добавить действие `move_right`.



Нам нужно назначить клавишу для этого действия. Нажмите на значок "+" справа, чтобы открыть окно менеджера событий.



Поле "Прослушивание ввода..." должно быть выбрано автоматически. Нажмите клавишу "вправо" на клавиатуре, и меню теперь должно выглядеть следующим образом.

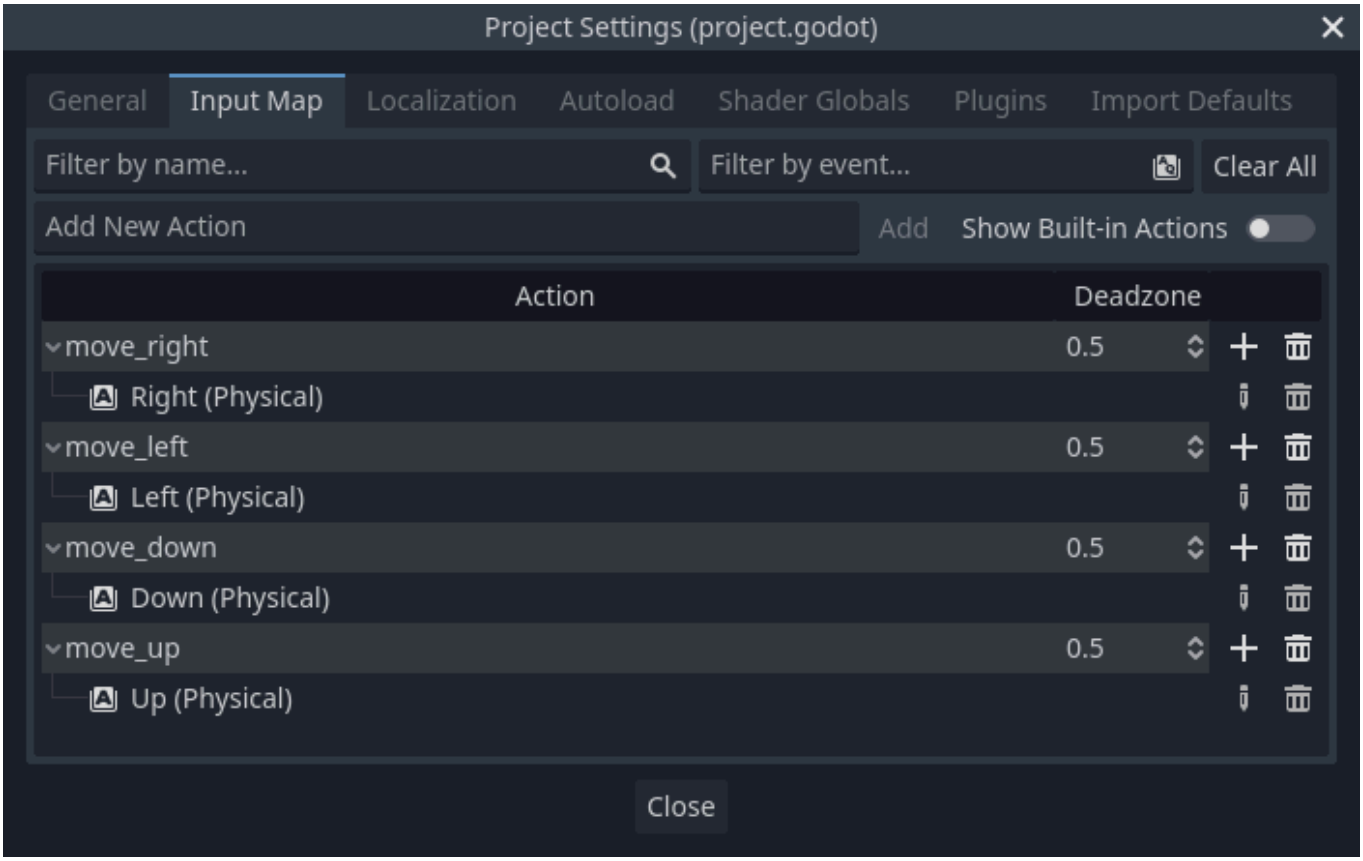


Нажмите кнопку "ок". Клавиша "вправо" теперь связана с действием `move_right`.

Повторите эти шаги, чтобы добавить ещё три маппинга:

1. `move_left` соответствует стрелке влево.
2. `move_up` соответствует стрелке вверх.
3. А `move_down` соответствует стрелке вниз.

Вкладка Список действий должна выглядеть так:



Нажмите кнопку "Закрыть" чтобы закрыть настройки проекта.

> Мы присвоили только одну клавишу каждому действию, но Вы можете присвоить тем же самым действиям несколько клавиш, кнопки джойстика или мыши.

Вы можете определить, нажата ли клавиша с помощью функции `Input.is_action_pressed()`, которая возвращает `true`, если клавиша нажата, или `false`, если нет.

```
func _process(delta):
        var velocity = Vector2.ZERO # The player's movement vector.
        if Input.is_action_pressed("move_right"):
                velocity.x += 1
        if Input.is_action_pressed("move_left"):
                velocity.x -= 1
        if Input.is_action_pressed("move_down"):
                velocity.y += 1
        if Input.is_action_pressed("move_up"):
                velocity.y -= 1

        if velocity.length() > 0:
                velocity = velocity.normalized() * speed
                $AnimatedSprite2D.play()
```

```
        else:
                $AnimatedSprite2D.stop()
```

Начнём с того, что установим значение `velocity` в `(0, 0)` - по умолчанию игрок двигаться не должен. Затем, мы проверяем каждый ввод и добавляем/вычитаем значение из `velocity`, чтобы получить общее направление. Например, если вы одновременно удерживаете `right` и `down`, полученный вектор `velocity` будет `(1, 1)`. В этом случае, поскольку мы добавляем горизонтальное и вертикальное движение, игрок будет двигаться *быстрее*, чем если бы он перемещался только по горизонтали.

Можно избежать этого, если мы *нормализуем* скорость, что означает, что мы устанавливаем ее *длину* на `1` и умножаем на желаемую скорость. Это означает отсутствие более быстрого диагонального движения.

> Если вы никогда раньше не использовали векторную математику или нуждаетесь в повторении, вы можете увидеть объяснение использования вектора в Godot по ссылке Векторная математика. Ее полезно знать, но она не понадобится для остальной части этого урока.

Мы также проверяем, движется ли игрок, чтобы мы могли вызвать `play()` или `stop()` в AnimatedSprite.

> `$` является сокращением для `get_node()`. В приведенном выше коде `$AnimatedSprite.play()` то же самое, что и `get_node("AnimatedSprite").play()`.
>
> В GDScript `$` возвращает узел по относительному пути от текущего узла или возвращает `null`, если узел не найден. Поскольку `AnimatedSprite2D` является дочерним элементом текущего узла, мы можем использовать `$AnimatedSprite2D`.

Теперь, когда у нас есть направление движения, мы можем обновить позицию игрока. Мы также можем использовать `clamp()`, чтобы он не покинул экран. *Clamping* означает ограничение движения диапазоном. Добавьте следующее в конец функции `_process` (убедитесь, что он не имеет отступа под else):

```
position += velocity * delta
position = position.clamp(Vector2.ZERO, screen_size)
```

> Параметр "delta" в функции "*process()" означает* длительность кадра_, то есть время, потраченное на завершение обработки предыдущего кадра. Благодаря использованию этого значения скорость движения будет постоянной даже при изменении частоты кадров.

Нажмите «Запустить текущую сцену» (F6) и убедитесь, что вы можете перемещать игрока по экрану во всех направлениях.

> Если вы получаете ошибку в панели "Отладчик", с надписью
>
> ``Attempt to call function 'play' in base 'null instance' on a null instance``
>
> это, скорее всего, означает, что вы ввели название узла AnimatedSprite неверно. Имена узлов чувствительны к регистру, а `$NodeName` должен совпадать с именем, которое вы видите в дереве сцены.

Выбор анимации

Теперь, когда игрок может двигаться, нам нужно изменять анимацию AnimatedSprite2D в зависимости от направления движения. У нас есть анимация "walk", которая показывает игрока, идущего направо. Эту анимацию следует перевернуть горизонтально, используя свойство `flip_h` для движения влево. У нас также есть анимация "up", которую нужно перевернуть вертикально с помощью `flip_v` для движения вниз. Поместим этот код в конец функции `_process()`:

```
if velocity.x != 0:
        $AnimatedSprite2D.animation = "walk"
        $AnimatedSprite2D.flip_v = false
        # See the note below about the following boolean assignment.
        $AnimatedSprite2D.flip_h = velocity.x < 0
elif velocity.y != 0:
        $AnimatedSprite2D.animation = "up"
        $AnimatedSprite2D.flip_v = velocity.y > 0
```

> Логические присваивания в коде выше являются общим сокращением для программистов. Поскольку мы проводим проверку сравнения (логическую, булеву), а также *присваиваем* булево значение, мы можем делать и то, и другое одновременно. Рассмотрим этот код в сравнении с однострочным логическим присваиванием выше:

```
if velocity.x < 0:
        $AnimatedSprite2D.flip_h = true
else:
        $AnimatedSprite2D.flip_h = false
```

Воспроизведите сцену еще раз и проверьте правильность анимации в каждом из направлений.

> Общей ошибкой является неправильное именование анимаций. Имена анимаций в панели SpriteFrames должны совпадать с именами анимаций в вашем коде. Если вы назвали анимацию "Walk", вы должны также использовать заглавную букву "W" в коде.

Общей ошибкой является неправильное именование анимаций. Имена анимаций в панели SpriteFrames должны совпадать с именами анимаций в вашем коде. Если вы назвали анимацию "Walk", вы должны также использовать заглавную букву "W" в коде.

Если вы уверены, что движение работает правильно, добавьте эту строку в `_ready()`, чтобы игрок был скрыт при запуске игры:

```
hide()
```

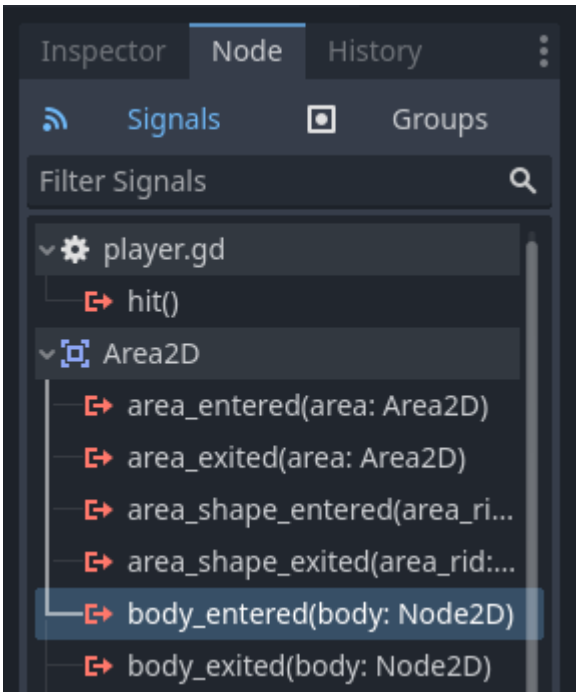
Подготовка к столкновениям

Мы хотим, чтобы `Player` обнаруживал столкновение с врагом, но мы еще не сделали никаких врагов! Это нормально, потому что мы будем использовать такой функционал Godot, как *сигнал*.

Добавьте следующее в верхней части скрипта. Если вы используете GDScript, добавьте его после `extends Area2D`. Если вы используете C#, добавьте его после `public partial class Player : Area2D`:

```
signal hit
```

Это определяет пользовательский сигнал под названием "hit" ("удар"), который наш игрок будет излучать (отправлять), когда он сталкивается с противником. Мы будем использовать `Area2D` для обнаружения столкновения. Выберите узел `Player` ("Игрок") и щелкните по вкладке "Узел" (Node) рядом с вкладкой "Инспектор" (Inspector), чтобы просмотреть список сигналов, которые игрок может посылать:



Обратите внимание, что наш пользовательский сигнал "hit" там также есть! Поскольку наши противники будут узлами `RigidBody2D`, нам нужен сигнал `body_entered(body: Node2D)`. Он будет отправляться при контакте тела (body) с игроком. Нажмите "Присоединить..." - появится окно "Подключить сигнал к методу".

Godot создаст для вас функцию с таким же точным именем прямо в скрипте. Пока что вам не нужно менять настройки по умолчанию.

```
8  func _on_body_entered(body):
9      pass # Replace with function body.
10
```

Обратите внимание на зеленый значок, указывающий на то, что сигнал подключен к этой функции; это не означает, что функция существует, а только то, что сигнал попытается подключиться к функции с таким именем, поэтому дважды проверьте, чтобы написание функции точно совпадало!

Затем добавьте этот код в функцию:

```
func _on_body_entered(body):
        hide() # Player disappears after being hit.
        hit.emit()
        # Must be deferred as we can't change physics properties on a physics
callback.
        $CollisionShape2D.set_deferred("disabled", true)
```

Каждый раз, когда противник ударяет игрока, будет посылаться сигнал. Нам нужно отключить столкновение игрока, чтобы не вызывать сигнал `hit` более одного раза.

> Отключение формы области столкновения может привести к ошибке, если это происходит во время обработки движком столкновений.
> Использование `set_deferred()` говорит Godot ждать отключения этой формы, пока это не будет безопасно.

Последняя деталь - добавить функцию, которую мы можем вызвать для перезагрузки игрока при запуске новой игры.

```
func start(pos):
        position = pos
        show()
        $CollisionShape2D.disabled = false
```

Создание врага

Теперь пришло время сделать врагов, от которых нашему игроку предстоит уклоняться. Их поведение будет не очень сложным: толпы (мобы) будут появляться случайно по краям экрана, выбирать случайное направление и двигаться по прямой линии.

Мы создадим сцену `Mob`, которую затем сможем *инстанцировать*, чтобы создать любое количество независимых мобов в игре.

Настройка узла

Нажмите "Сцена" -> "Новая сцена" и добавьте следующие узлы:

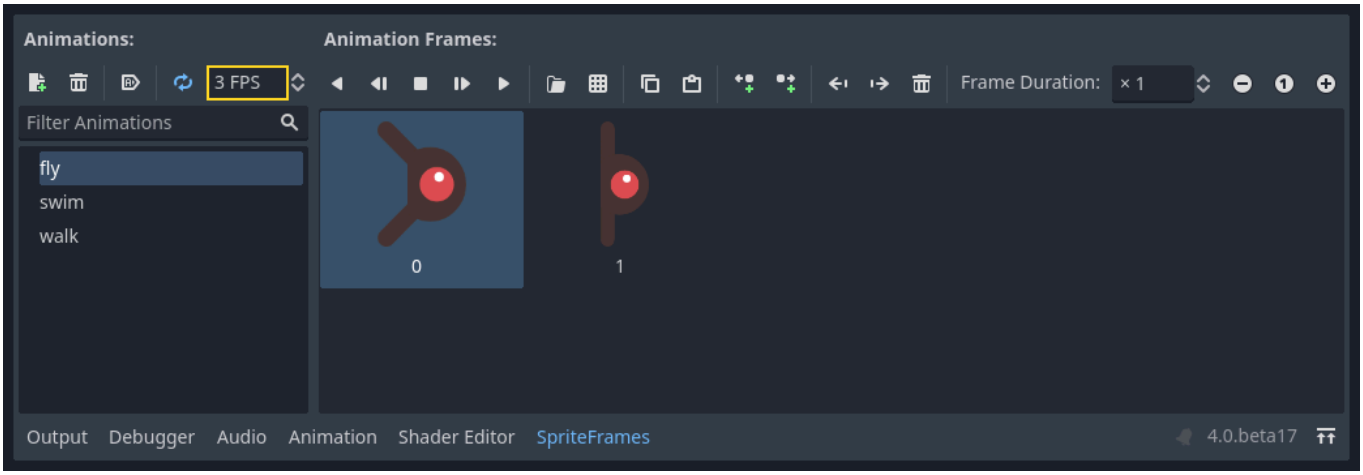
- RigidBody2D (названный `Mob`)
- AnimatedSprite2D
- CollisionShape2D
- VisibleOnScreenNotifier2D

Не забудьте настроить дочерние узлы, чтобы они не могли быть выбраны, как вы это делали со сценой игрока.
Выберите узел `Mob` и установите его свойство `Gravity Scale` в разделе `RigidBody2D` инспектора на 0. Это предотвратит падение моба вниз.

Кроме того, в разделе `CollisionObject2D`, расположенном сразу под разделом **RigidBody2D**, разверните группу **Collision** и выключите слой `1` в разделе `Mask`. Это гарантирует, что мобы не столкнутся друг с другом.

Настройте AnimatedSprite2D, как вы сделали это для игрока. На этот раз у нас есть 3 анимации: `fly` (лететь), `swim` (плыть) и `walk` (ходить). В папке art есть две картинки для каждой анимации.

`Скорость анимации` должна быть установлена для каждой анимации отдельно. Установите значение `3` для всех трех анимаций.



Вы можете использовать кнопку "Воспроизвести анимацию" справа от поля ввода `Скорость анимации` для предварительного просмотра ваших анимаций.

Мы выберем одну из этих анимаций случайным образом, так что мобы будут иметь некоторое разнообразие.

Также как изображения игрока, изображения мобов нужно уменьшить. Установите для `AnimatedSprite2D` в свойстве `Scale` значение `(0.75, 0.75)`.

Как и в сцене `Player`, добавьте форму `CapsuleShape2D` для столкновений. Чтобы выровнять фигуру с изображением, вам нужно установить свойство `Rotation` на значение `90` (в разделе "Transform" в инспекторе).

Сохраните сцену.

Скрипт врага

Добавьте этот скрипт к `Mob`:

```
extends RigidBody2D
```

Теперь давайте посмотрим на остальную часть скрипта. В функции `_ready()` мы проигрываем анимацию и случайным образом выбираем один из трех типов анимации:

```
func _ready():
	var mob_types = $AnimatedSprite2D.sprite_frames.get_animation_names()
	$AnimatedSprite2D.play(mob_types[randi() % mob_types.size()])
```

Сначала мы получаем список названий анимаций из свойства AnimatedSprite `frames`. Возвращается Массив, содержащий все три названия анимации: `["walk", "swim", "fly"]`.

Дальше нам нужно выбрать случайное число между `0` и `2`, чтобы выбрать имя одной из анимаций из списка (индексы массива начинаются с `0`). Функция `randi() % n` выбирает случайное целое число между `0` и `n-1`.

Последняя часть - заставить мобов удаляться, когда они выходят за экран. Подключите сигнал `screen exited()` узла `Visibility Notifier 2D` и добавьте этот код:

```
func _on_visible_on_screen_notifier_2d_screen_exited():
	queue_free()
```

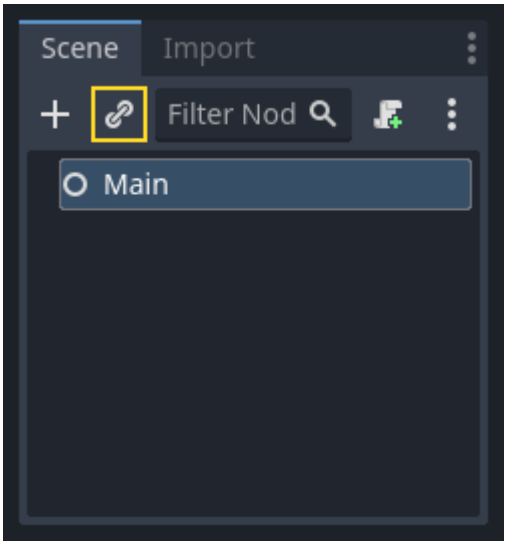
На этом заканчиваем работу над сценой Mob.
Теперь, когда игрок и враги готовы, в следующей части мы перенесём их на новую сцену. Мы заставим врагов появляться случайно по границам экрана и двигаться вперёд, сделав наш проект играбельным.

Главная сцена игры

Итак, настало время перенести всё, что мы сделали вместе, на играбельную игровую сцену.

Создайте новую сцену и добавьте Node с именем `Main`. (Причиной, по которой мы используем Node, а не Node2D, является то, что узел будет контейнером для обработки игровой логики. Это не требует именно двумерного функционала.)

Нажмите на кнопку **Instance** (**Экземпляр**, выглядящий как значок звена цепи) и выберите ваш сохраненный `player.tscn`.



Теперь добавьте следующие узлы в качестве дочерних узлов `Main` и назовите их, как показано:

- Timer (названный `MobTimer`) - чтобы контролировать частоту появления мобов
- Timer (названный `ScoreTimer`) - чтобы каждую секунду увеличивать счет
- Timer (названный `StartTimer`) - чтобы дать задержку перед стартом игры
- Marker2D (названный `StartPosition`) - чтобы указать начальную позицию игрока
  Установите свойство `Wait Time` каждого узла `Timer` следующим образом (значения указаны в секундах):
- `MobTimer`: `0.5`
- `ScoreTimer`: `1`
- `StartTimer`: `2`
  Кроме того, установите для свойства `One Shot` узла `StartTimer` значение "Вкл" и для свойства `Position` узла `StartPosition` установите значение `(240, 450)`.

Добавление мобов

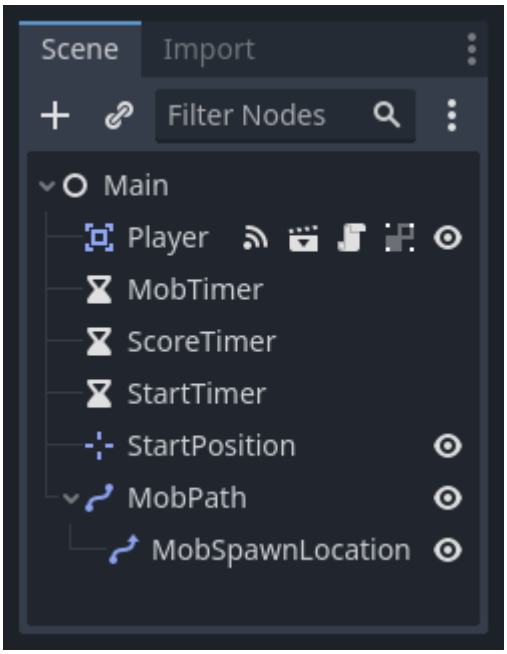
Узел Main будет порождать новых мобов, и мы хотим, чтобы они появлялись в случайном месте на краю экрана. Добавьте узел Path2D с именем `MobPath` как дочерний элемент узла `Main`. Когда вы выберете `Path2D`, вы увидите несколько новых кнопок в верхней части редактора:

Выберите среднюю ("Добавить точку") и нарисуйте путь щелчками мыши, чтобы добавить точки в показанных углах. Чтобы точки привязывались к сетке, убедитесь, что выбраны "Использовать привязку к сетке" и "Использовать умную привязку". Эти опции можно

найти слева от кнопки "Заблокировать узел", они отображаются в виде магнита с точками и магнита с пересекающимися линиями.

> Нарисуйте путь в порядке *по часовой стрелке*, иначе ваши мобы будут появляться с направлением *наружу*, а не *внутрь*!

Поместив точку "4" на изображение, нажмите кнопку "Сомкнуть кривую", и она будет завершена.
Теперь, когда путь определен, добавьте узел PathFollow2D как дочерний элемент `MobPath` и назовите его `MobSpawnLocation`. Этот узел будет автоматически вращаться и следовать по пути при его перемещении, поэтому мы можем использовать его для выбора случайной позиции и направления вдоль пути.
Ваша сцена должна выглядеть так:



Главный скрипт

Добавьте скрипт к `Main`. В верхней части скрипта мы пишем `@export var mob_scene: PackedScene`, что позволяет нам выбрать сцену Mob, экземпляр которой мы хотим сделать.

```
extends Node

@export var mob_scene: PackedScene
var score
```

Щелкните узел `Main`, и вы увидите свойство `Mob Scene` в инспекторе под «`Main.gd`». Значение этого свойства можно присвоить двумя способами:

- Перетащите `Mob.tscn` из панели "Файловая система" в свойство **Mob Scene**.
- Нажмите стрелочку вниз рядом с "[пусто]" и выберите "Загрузить" ("Load"). Затем выберите `mob.tscn`.
  Затем выберите экземпляр сцены «Player» в узле «Main» в панели «Сцена» и откройте панель Узла на боковой панели. Убедитесь, что в панели Узла выбрана вкладка Сигналы(Signals).
  Вы должны увидеть список сигналов для узла `Player`. В списке найдите и дважды щелкните по сигналу `hit` (или щелкните по нему правой кнопкой мыши и выберите "Присоединить..."). Это откроет диалоговое окно подключения сигнала. Мы хотим создать новую функцию с именем `game_over`, которая будет обрабатывать то, что должно произойти, когда игра заканчивается. Введите "game_over" в поле "Receiver Method"("Метод получения") в нижней части диалогового окна подключения сигнала и нажмите "Присоединить". Добавьте следующий код в новую функцию, а также функцию `new_game`, которая настроит всё для новой игры:

```
func game_over():
        $ScoreTimer.stop()
        $MobTimer.stop()

func new_game():
        score = 0
        $Player.start($StartPosition.position)
        $StartTimer.start()
```

Теперь присоедините сигнал `timeout()` каждого из узлов Timer (`StartTimer`, `ScoreTimer` и `MobTimer`) к главному скрипту. `StartTimer` запустит два других таймера. `ScoreTimer` будет увеличивать счет на 1.

```
func _on_score_timer_timeout():
        score += 1

func _on_start_timer_timeout():
        $MobTimer.start()
        $ScoreTimer.start()
```

В функции `_on_mob_timer_timeout()` мы создадим экземпляр моба, выберем случайное начальное местоположение вдоль `Path2D` и приведем его в движение.
Узел `PathFollow2D` будет автоматически поворачивать его по направлению пути, поэтому мы воспользуемся этим, чтобы выбрать направление моба и его позицию. Когда мы

создаем моба, получаем случайное значение от `150.0` до `250.0` - это определяет скорость движения моба (было бы скучно, если бы они все двигались с одинаковой скоростью).

Обратите внимание, что новый экземпляр должен быть добавлен в сцену с помощью функции `add_child()`.

```
func _on_mob_timer_timeout():
        # Create a new instance of the Mob scene.
        var mob = mob_scene.instantiate()

        # Choose a random location on Path2D.
        var mob_spawn_location = $MobPath/MobSpawnLocation
        mob_spawn_location.progress_ratio = randf()

        # Set the mob's direction perpendicular to the path direction.
        var direction = mob_spawn_location.rotation + PI / 2

        # Set the mob's position to a random location.
        mob.position = mob_spawn_location.position

        # Add some randomness to the direction.
        direction += randf_range(-PI / 4, PI / 4)
        mob.rotation = direction

        # Choose the velocity for the mob.
        var velocity = Vector2(randf_range(150.0, 250.0), 0.0)
        mob.linear_velocity = velocity.rotated(direction)

        # Spawn the mob by adding it to the Main scene.
        add_child(mob)
```

> Почему `PI`? В функциях, требующих углы, Godot использует *радианы*, а не градусы. Число Пи представляет собой пол-оборота в радианах, примерно `3.1415` (также есть переменная `TAU`, которая равна `2 * PI`) Если вам удобнее работать с градусами, вам нужно использовать функции `deg2rad()` и `rad2deg()`.

Тестирование сцены

Давайте протестируем сцену, чтобы убедиться, что все работает. Добавьте в `_ready()` вызов `new_game`:

```
func _ready():
```

```
        new_game()
```

Также давайте назначим сцену `Main` в качестве нашей "Главной сцены", которая запускается автоматически при запуске игры. Нажмите кнопку "Запустить проект" и выберите `main.tscn` при появлении запроса.

> Если вы уже установили другую сцену в качестве «Основной сцены», вы можете щелкнуть правой кнопкой мыши по `main.tscn` в панели «Файловая система» и выбрать «Установить как основную сцену».

У вас должна быть возможность перемещать игрока, видеть, как появляются мобы, и видеть, как игрок исчезает, когда его бьет моб.
Когда вы убедитесь, что все работает, удалите вызов `new_game()` из `_ready()` и замените его на `pass`.

Графический интерфейс

Заключительный этап нашей игры нуждается в Пользовательском интерфейсе (UI), чтобы отображать такие вещи, как рейтинг, сообщение "игра окончена" и кнопку перезапуска.

Создайте новую сцену и добавьте узел `CanvasLayer` с именем `HUD`. "HUD" означает "heads-up display" ("графический интерфейс пользователя"), дисплей, отображающийся поверх игры.

Узел `CanvasLayer` позволяет нам прорисовывать элементы нашего UI на слое поверх всей остальной игры, поэтому отображаемая информация не перекрывается никакими игровыми элементами, такими как игрок или мобы.
HUD должен отображать следующую информацию:

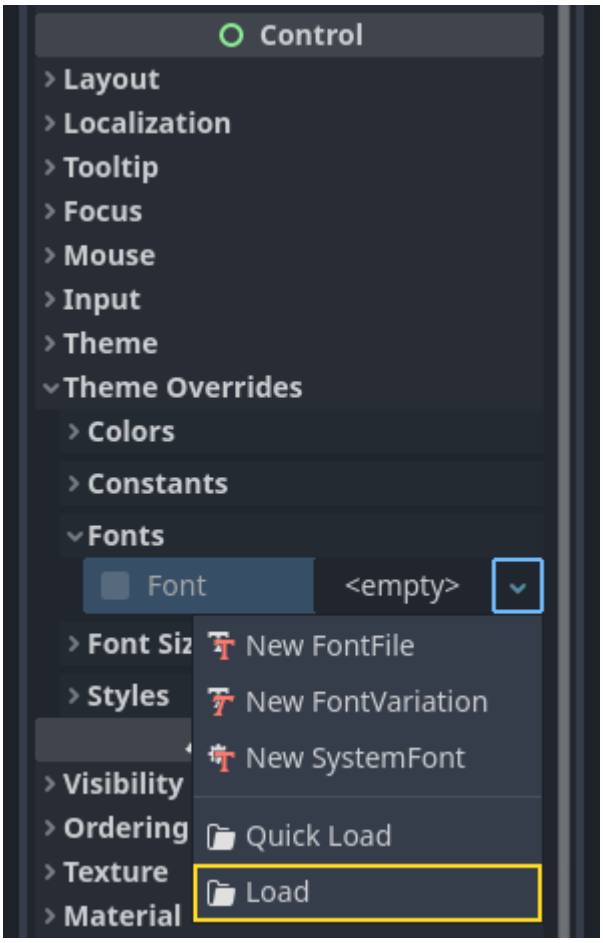
- Счет, измененный `ScoreTimer`.
- Сообщение, например "Game Over" или "Get Ready!"
- Кнопка "Start", чтобы начать игру.
  Основной узел для элементов UI — это Control. Чтобы создать наш UI, мы будем использовать два типа узлов Control: Label и Button.

Создайте следующие узлы, как дочерние узла `HUD`:

- Label с именем `ScoreLabel`.
- Label с именем `Message`.
- Button с именем `StartButton`.
- Timer с именем `MessageTimer`.
  Нажмите на `ScoreLabel` и введите число в поле `Text` в Инспекторе. Стандартный

шрифт для узлов `Control` мал и плохо масштабируется. В ресурсы игры включен файл шрифта под названием "Xolonium-Regular.ttf". Чтобы использовать этот шрифт, сделайте следующее:

В "Theme Overrides > Fonts" выберите "Загрузить" и выберите файл "Xolonium-Regular.ttf".

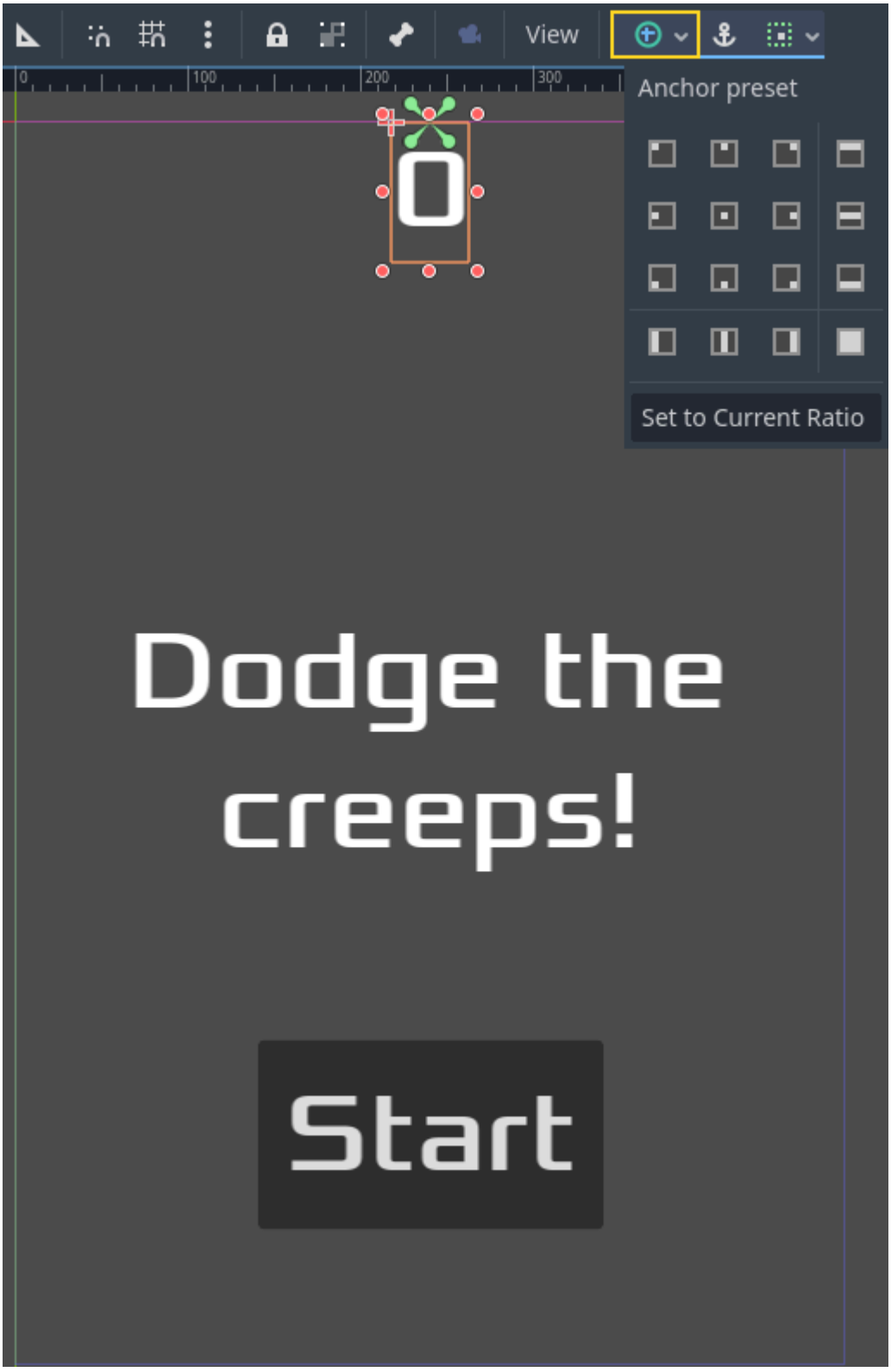


Размер шрифта все еще слишком мал, увеличьте его до `64` в разделе «Theme Overrides > Font Sizes». Сделав это для `ScoreLabel`, повторите изменения для узлов `Message` и `StartButton`.

> Якоря: Управляющие узлы имеют позицию и размер, но у них также есть якоря. Якоря определяют начало координат — опорную точку для краев узла.

Расположите узлы, как показано ниже. Вы можете перетаскивать узлы, чтобы разместить их вручную, или для более точного размещения используйте "Пресеты для значения

якорей".



ScoreLabel

1. Установите текст на `0`.
2. В Инспекторе, установите "Horizontal Alignment" и "Vertical Alignment" на `Center`.
3. Установите "пресет якорей" на `Вверху посередине`.

Message

1. Установите текст на `Уверись от Крипов!`.
2. В Инспекторе, установите "Horizontal Alignment" и "Vertical Alignment" на `Center`.
3. Установите "Autowrap Mode" на `Word`, в противном случае надпись останется на одной строке.
4. В разделе "Control - Layout/Transform" установите для параметра «Size X» значение `480`, чтобы использовать всю ширину экрана.
5. Установите "Пресет якорей" на `Центр`.

StartButton

1. Установите текст на `Старт`.
2. В разделе "Control - Layout/Transform" установите для параметра «Size X» значение `480`, а для параметра "Size Y" значение `100`, чтобы добавить немного больше отступа между границей экрана и текстом.
3. Установите "Пресет якорей " на `Внизу посередине`.
4. В разделе "Control - Layout/Transform" установите для параметра "Position Y" значение `580`.
   В `MessageTimer` установите параметр `Wait Time` на `2`, а параметр `One Shot` на значение "Вкл".

Теперь добавьте этот скрипт в `HUD`:

```
extends CanvasLayer

# Notifies `Main` node that the button has been pressed
signal start_game
```

Теперь мы хотим временно отобразить сообщение, например "Get Ready" (Приготовьтесь), поэтому мы добавляем следующий код

```
func show_message(text):
        $Message.text = text
        $Message.show()
        $MessageTimer.start()
```

Также нам нужна функция, которая будет вызываться, когда игрок проигрывает. Она покажет надпись "Game Over" на 2 секунды, затем произойдет возврат к основному экрану, и после короткой паузы появится кнопка "Старт".

```
func show_game_over():
        show_message("Game Over")
        # Wait until the MessageTimer has counted down.
        await $MessageTimer.timeout

        $Message.text = "Dodge the Creeps!"
        $Message.show()
        # Make a one-shot timer and wait for it to finish.
        await get_tree().create_timer(1.0).timeout
        $StartButton.show()
```

Эта функция вызывается, когда игрок проигрывает. Она покажет надпись "Game Over" на 2 секунды, затем произойдет возврат к основному экрану, и после короткой паузы появится кнопка "Start".

> Если вам нужно сделать паузу на короткое время, то альтернативой использованию узла Timer является использование функции SceneTree `create_timer()`. Может быть очень полезно добавлять задержки наподобие таких, как в вышеприведенном коде, где нам хотелось бы подождать немного времени, прежде чем показывать кнопку "Start".

Добавьте приведенный ниже код в `HUD`, чтобы обновлять счет

```
func update_score(score):
        $ScoreLabel.text = str(score)
```

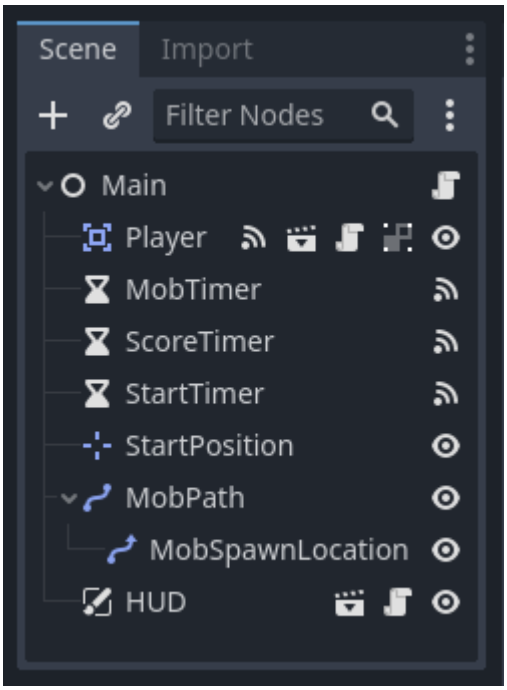
Подключите сигнал `press()` `StartButton` и сигнал `timeout()` `MessageTimer` к узлу `HUD` и добавьте следующий код к новым функциям:

```
func _on_start_button_pressed():
        $StartButton.hide()
        start_game.emit()

func _on_message_timer_timeout():
        $Message.hide()
```

Подключение HUD к Main

Теперь, когда мы закончили создание сцены `HUD`, вернитесь к `Main`. Инстанцируйте сцену `HUD` в `Main` подобно тому, как вы это делали со сценой `Player`. Дерево сцены должно выглядеть так, поэтому убедитесь, что вы ничего не упустили:



Теперь нам нужно подключить функционал `HUD` в наш `Main`-скрипт. Для этого потребуются некоторые дополнения к сцене `Main`:

Во вкладке "Узел" присоедините сигнал HUD `start_game` к функции узла Main `new_game()`, введя "new_game" в поле "Принимающий метод" в окне "Присоединить сигнал к методу" . Убедитесь, что в скрипте рядом с функцией `func new_game()` теперь появилась зелёная иконка подключения.

В `new_game()` обновим отображение счёта и выведем сообщение "Get Ready":

```
$HUD.update_score(score)
$HUD.show_message("Get Ready")
```

В `game_over()` нам нужно вызвать соответствующую функцию `HUD`:

```
$HUD.show_game_over()
```

Наконец добавьте этот код в `_on_score_Ttmer_timeout()`, чтобы синхронизировать отображение с изменением количества очков:

```
$HUD.update_score(score)
```

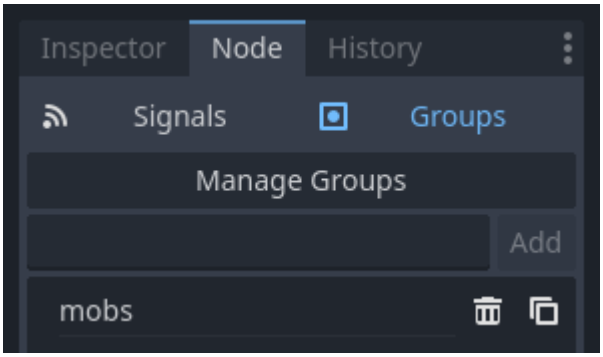
> Когда вы убедитесь, что всё работает, удалите вызов `new_game()` из `_ready()`, иначе ваша игра запустится автоматически.

Теперь можно поиграть! Нажмите на кнопку "Запустить проект". Вам будет предложено выбрать основную сцену, выбирайте `main.tscn`.

Удаляем старых крипов

Если вы играете до "Game Over", а затем сразу начинаете новую игру, то крипы из предыдущей игры могут все еще оставаться на экране. Было бы лучше, если бы все они исчезали в начале новой игры. Нам просто нужен способ сказать *всем* мобам, чтобы они удалились. Мы можем сделать это с помощью функции "group" ("группа").

В сцене `Mob` выберите корневой узел и нажмите вкладку "Узел" рядом с Инспектором (там же, где вы находите сигналы узла). Рядом с "Сигналы" нажмите "Группы", введите новое имя группы и нажмите "Добавить".



Теперь все мобы будут в группе "mobs". Затем мы можем добавить следующие строки к функции `new_game()` в `Main`:

```
get_tree().call_group("mobs", "queue_free")
```

Функция `call_group()` вызывает каждую именованную функцию на каждом узле в группе - в этом случае мы говорим каждому мобу удалять себя.

Завершающие штрихи

На данный момент мы завершили всю функциональность для нашей игры. Ниже остаются некоторые шаги, слегка добавляющие "сока" и улучшающие игровой опыт.

Не стесняйтесь совершенствовать геймплей своими собственными идеями.

Фон

Серый фон по умолчанию не очень привлекателен, так что давайте поменяем его цвет. Один из способов сделать это - использовать узел ColorRect. Создайте его первым узлом под `Main`, таким образом он будет прорисовываться за другими узлами. `ColorRect` имеет только одно свойство: `Color`. Подберите цвет, который вам нравится, и выберите "Layout"

-> "Anchors Preset" -> "Полный прямоугольник" либо на панели инструментов в верхней части окна просмотра, либо в инспекторе так, чтобы он закрывал экран.

Вы также можете добавить фоновое изображение, если оно у вас есть, используя вместо этого узел `TextureRect`.

Звуковые эффекты

Звук и музыка могут быть одним из самых эффективных способов придания игровому процессу большей привлекательности. В папке ресурсов вашей игры есть два звуковых файла: "House In a Forest Loop.ogg" для фоновой музыки и "gameover.wav" для проигрыша игрока.

Добавьте два узла AudioStreamPlayer как дочерние для `Main`. Назовите один из них `Music`, а другой - `DeathSound`. В каждом из них нажмите на свойство `Stream`, выберите "Загрузить" и добавьте соответствующий звуковой файл.

Весь звук автоматически импортируется при отключенном параметре `Loop` (зацикливание). Если вы хотите, чтобы музыка зациклилась, щелкните маленькую стрелку справа от параметра Stream, выберите `Сделать уникальным`, затем нажмите на файл Stream и установите `Loop` (с англ. - зациклить).

Для воспроизведения музыки добавьте `$Music.play()` в функцию `new_game()` и `$Music.stop()` в функцию `game_over()`.

Наконец, добавьте `$DeathSound.play()` в функцию `game_over()`.

```
func game_over():
        ...
        $Music.stop()
        $DeathSound.play()

func new_game():
        ...
        $Music.play()
```

Сочетание клавиш

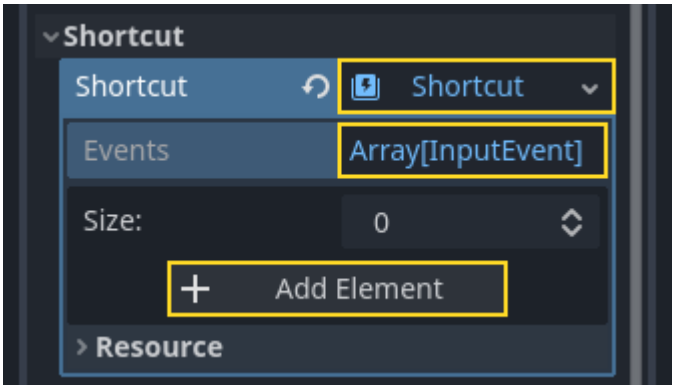
Поскольку в игру играют с помощью клавиш управления, было бы удобно, если бы мы могли начать игру, нажав клавишу на клавиатуре. Мы можем сделать это с помощью свойства "Shortcut" узла `Button`.

В предыдущем уроке мы создали четыре события ввода для перемещения персонажа. Мы создадим аналогичное событие ввода для привязки к кнопке "Старт".

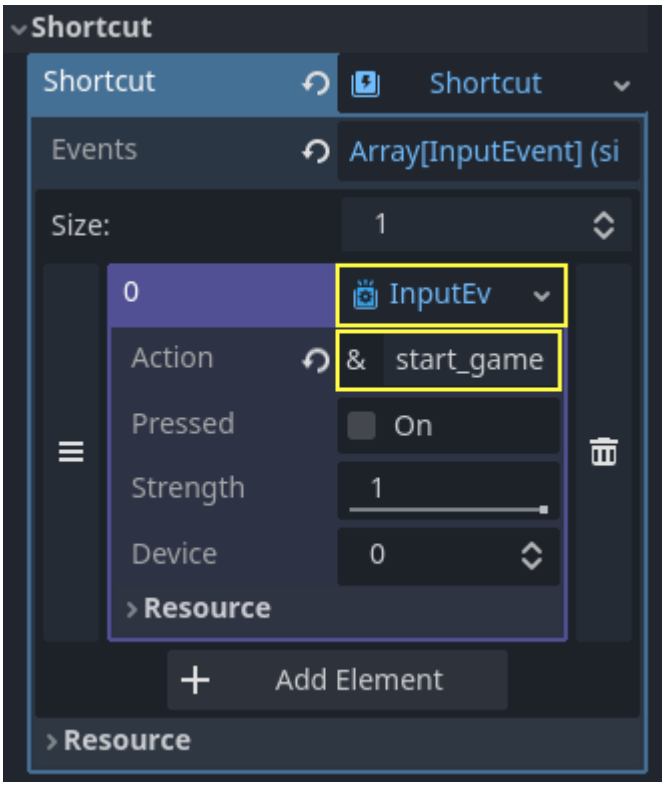
Выберите "Проект" -> "Настройки проекта", а затем перейдите на вкладку "Список действий". Точно так же, как вы создавали события ввода для движения, создайте новое событие под названием `start_game` и добавьте в него клавишу Enter`.

Сейчас самое время добавить поддержку контроллера, если он у вас есть. Подключите контроллер, а затем под каждым действием ввода, для которого вы хотите добавить поддержку контроллера, нажмите кнопку "+" и нажмите соответствующую кнопку, крестовину или направление джойстика, которое вы хотите сопоставить с соответствующим действием ввода. .

В сцене `HUD` выберите кнопку `StartButton` и найдите ее свойство Shortcut в инспекторе. Создайте новый ресурс Shortcut, щелкнув внутри поля, откройте массив «Events» и добавьте к нему новый элемент массива, щелкнув **Array[InputEvent] (size 0)**.



Создайте новое `InputEventAction` и назовите его `start_game`.



Теперь, когда появляется кнопка запуска, вы можете либо нажать на неё, либо на клавишу Enter, чтобы начать игру.

И на этом вы завершили свою первую 2D-игру в Godot.
Вы сделали персонажа, управляемого игроком, врагов, случайно появляющихся на игровом поле, подсчёт очков, реализовали завершение и повтор игры, пользовательский интерфейс, звуки и многое другое. Поздравляем!

Вам еще многому предстоит научиться, но вы можете воспользоваться моментом, чтобы оценить то, чего вы достигли.